3.0 Development Process Overview

This chapter describes the development process in detail. A powerful feature of the overall development process is the concept of automated integration. This means that automated tools are used to combine and load segments, make environmental modifications requested by segments, make newly loaded segments available to authorized users, and identify places where segments conflict with each other. Traditional system level integration then becomes primarily a task of loading and testing segments. Traditional integration tasks are pushed as far down to the developer level as possible.

Prior to submitting a component to DISA, a developer must

- ¥ package the component as a segment,
- ¥ demonstrate COE compliance through tools and checklists,
- ¥ test the segment in isolation with the COE,
- ¥ provide required segment documentation, and
- ¥ demonstrate the segment operating within the COE.

The Software Support Activity (SSA) enters the segment into the on-line library for configuration management purposes and confirms COE compliance by running the same suite of tools as the developer. The SSA then tests interaction between segments and the impact on performance, memory utilization, etc. Since segments typically can only interact through the COE, the task is greatly simplified and the need for human intervention in the process is minimized.

An automated integration approach is a practical necessity. Not only are segments contributed by different services and agencies, but individual segments are created by a large body of different developers. Traditional integration approaches rapidly break down with the need to communicate to such a large number of people while the costs incurred to resolve inter-module conflicts at system integration time become prohibitive.

This chapter begins with a consistent approach to version numbering, followed by a detailed look at the development phases. The chapter ends with some special considerations for how to migrate legacy systems rather than developing from scratch. Because of the special importance of the on-line library, Chapter 7 is devoted to it and its features. For the present chapter, it is sufficient to note that there is a configuration management repository for segments.

Integration and testing of a segment within the COE is the responsibility of the segment developer. Verification of COE-compliance, integration of the system as a whole, and interoperability testing are performed by government integrators.

3.1 Version Numbering

The COE concept requires the ability for segments to state dependencies upon other segments. Dependencies may exist because one segment requires that another segment also be loaded, or that two segments conflict and can not both be present in the system at the same time. A third type of dependency can exist because segments are sometimes version dependent, and it must be possible to make meaningful comparisons of successive segment releases. A consistent approach to version numbering is thus a mandated feature of the COE standard. Version numbers are applied to all segments and all segment patches.

COE-based systems consist of a collection of segments. Each segment has its own individual version number. When a version number is applied to a COE-based system as a whole, the version number refers to the COE, not the version for each individual mission application or segment. While this may seem confusing, it is consistent with commercial practice. For example, one refers to the version of Microsoft Windows (analogous to the DII COE) as well as individual applications like Word or Excel (analogous to mission applications like GSORTS, or to COTS products like Netscape).

COE compliance mandates adherence to the version numbering scheme outlined in this section. Version number digits are frequently tied to the signature level required to authorize a product release. Hence they have programmatic importance as well as technical importance for distinguishing between segment upgrades.

3.1.1 Segment Version Numbers

Segment version numbers consist of a sequence of 4 digits, separated by decimal points, of the form

a.b.c.d

where each of the digits have a specific meaning. The first digit is a *major release* number and indicates a significant change in the architecture or operation of the segment. Backwards compatibility is not guaranteed when the major release number is incremented. The second digit indicates a *minor release* in which new features are added to the segment, but the fundamental segment architecture remains unchanged. A minor release may necessitate relinking to take advantage of updated API libraries, but APIs are preserved at the source code level except possibly on a documented and approved case-by-case basis. The third digit is a *maintenance release* number. New features may be added to the segment, but the emphasis is on optimizations, feature enhancements, or modifications to improve stability and useability. APIs are preserved and do not generally

require segments to recompile or relink during successive releases. The fourth digit is a *developer release* number.

The first three digits are assigned by DISA, but the final digit is reserved for developers. The fourth digit is updated to keep track of successive releases during the integration process..

Version number digits are incremented to indicate later releases of a segment. This scheme provides a readily apparent method for comparing successive releases of a segment. For example, a segment with version number 2.1.6.1 is a newer version than 2.1.0.5. Moreover, according to the scheme outlined, APIs are preserved. Segments using version 2.1.0.5 can usually be expected to work without any modification when loaded on a system using the 2.1.6.1 version.

When specifying version dependencies, this scheme also allows segments to indicate the degree to which they are version sensitive. For example, suppose Segment A requires use of Segment B. Segment A may indicate that it requires Segment B, version 2.3 indicating that any maintenance release of version 2.3 (e.g., 2.3.2.0, 2.3.1.2) is acceptable. The same approach works for specifying segment conflicts.

> It is a violation of the COE to fail to increment version numbers between subsequent segment releases. This applies to *all* segments whether they are COTS segments, COE component segments, or mission application segments.

3.1.2 COTS Version Numbers

COTS products will typically already have version numbers assigned to them, but the convention used is vendor specific. This makes it difficult to make meaningful version comparisons in the same sense as the previous subsection. A further complication is that COTS products must often be configured before they can be properly utilized in a COE-based system. For this reason, COTS segments are also assigned version numbers.

A COTS version number consists of a *primary* and *secondary* version number separated by the '/' character. The primary version number follows the same convention described in the previous subsection, while the secondary version number is the version number assigned by the vendor. Comparisons and dependency specifications are always performed using *only* the primary version number.

For example, the DII COE requires an increase in the amount of shared memory configured in the vendor supplied Solaris 2.3 Unix Operating System. A primary version number, such as 2.1.3.6, is assigned so that the operating system is referred to as version 2.1.3.6/SOL-2.3. Similarly, the X11R5 version of an X Windows server might have a version number assigned such as 2.3.0.4/X11R5.

COE-based systems are presently composed of segments contributed from ongoing programs which may already have an established convention for version numbering. A secondary version number may also be attached to such segments. As with COTS segments, only the primary version number is used in the COE.

3.1.3 Patch Version Numbers

Patches are indicated by appending the letter 'P' and a single number to the primary version number. For example, patch 12 to version 2.1.3.5 of a segment would be designated as version 2.1.3.5P12. Patch 4 to the Solaris Operating System example in the previous subsection would be designated as 2.1.3.6P4/SOL-2.3.

3.2 Process Flowchart

Figure 3-1 is a detailed flowchart of the development process, beginning with registering a segment to be developed and ending with ultimately installing the segment at an operational site. The major development phases are delineated by dashed lines in the figure and correspond to the subsections which follow. This process flow is the same for all segments, including patch segments. As can be seen, the process is indeed largely automated.

By necessity, the figure is abbreviated and does not show several key elements of the development process such as error tracking and reporting, a configuration control board, DISA architecture groups, or configuration management and quality assurance. Each of the elements is strongly implied by Figure 3-1, but their description is beyond the scope of this document.

At several places in Figure 3-1, segments are added to the on-line library, COE Software Repository System (CSRS). Segments are compressed and encrypted within the CSRS to reduce disk space and for added security. Segments are also encrypted and compressed when they are transmitted electronically across the network. These actions are performed automatically and are transparent to the user.

While electronic transmission of segments across the network is the preferred approach, it is not possible in certain cases. It is not practical to transmit the operating system, X Windows, or Motif across the network due to licensing restrictions and their size. Other segments, especially database segments, may be too large to send electronically or may have a security classification that requires special handling and tracking. Figure 3-1 should be understood with this in mind. Electronic transfer is performed when feasible, but an alternate route using tape or other media is used as well when required.

Figure 3-1 also shows several places, especially in the Segment Integration phase, where a "Notify" action occurs. This is an electronic notification of status to the segment developer, to the development community, or to the user community. The subsections below describe notifications in more detail, but obviously notifications of status are sent only to the cognizant parties, not necessarily to the entire community. Notification is accomplished by email, World-Wide-Web, newsgroups, and "paper" as appropriate.

The very nature of COE-based systems dictates that security measures be taken to prevent unauthorized disclosure or access to sensitive information, including project status or system problem reports. For this reason, access to software and project information is divided between Internet and SIPRNET with firewalls to

restrict access. This level of detail is not necessary for the overview presented in this chapter and has been omitted from Figure 3-1.

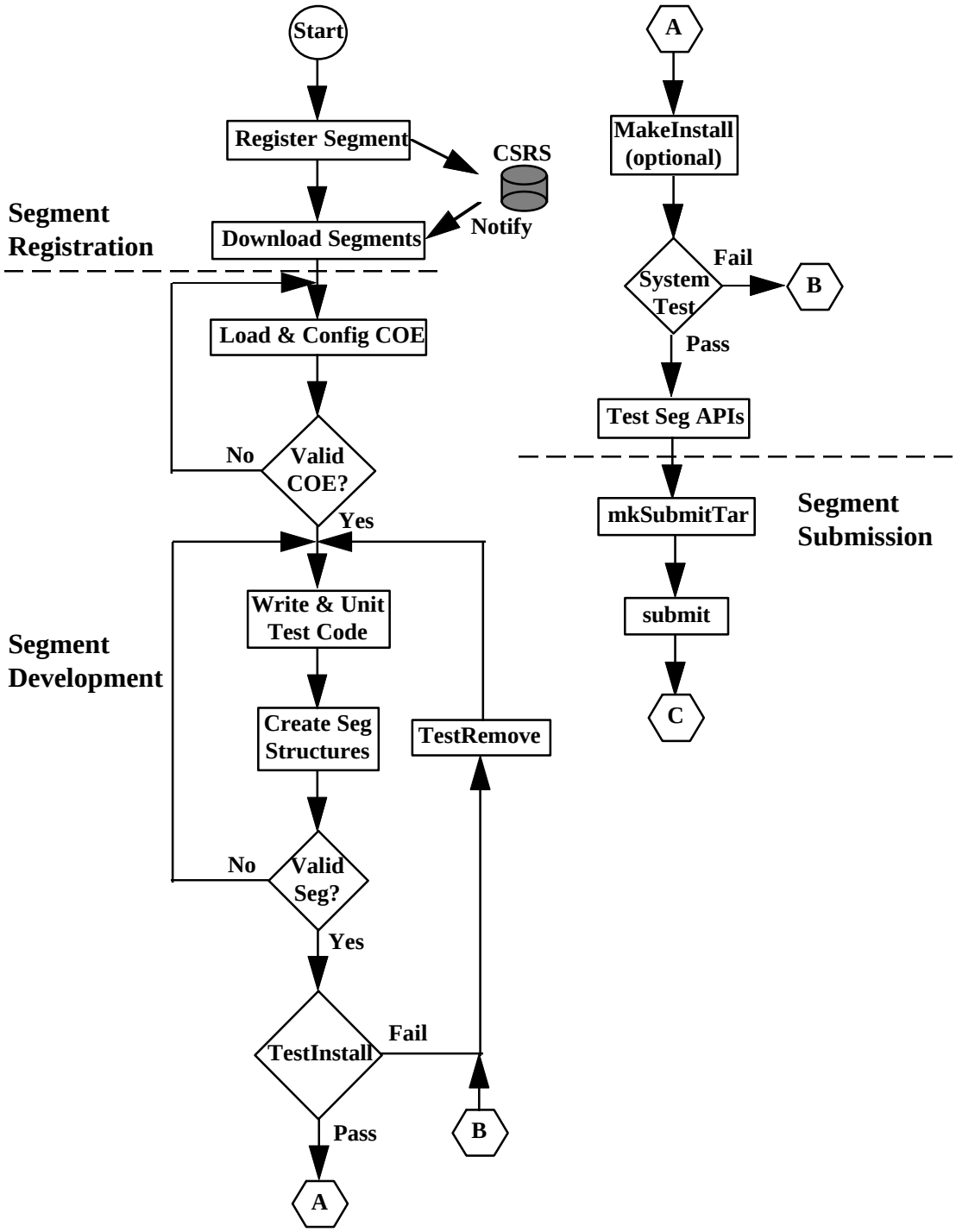

Figure 3-1: Development Process Overview

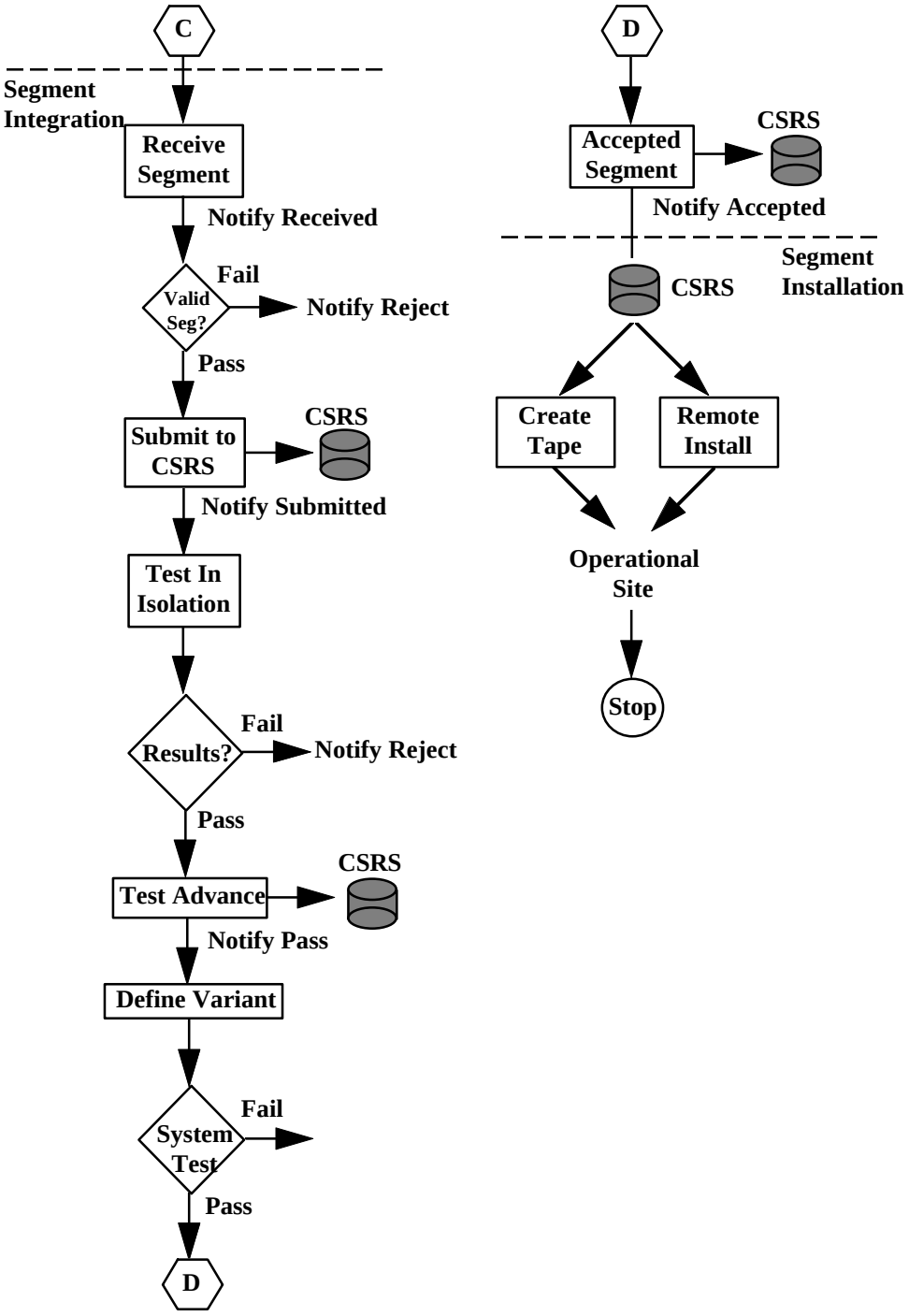

Figure 3-1: Development Process Overview (cont.)

3.2.1 Segment Registration

Segment Registration is the entry point into the development process. It's purpose is to collect information about the segment for publication in a *segment catalog*. Perhaps the most difficult part of maintaining a software repository is simply knowing what capabilities exist. This is the purpose of maintaining a DII segment catalog. The segment catalog is available on-line through an HTML browser and contains information provided by developers in a segment registration form. Keyword searches can be performed on the catalog by developers to identify reusable segments, or by operational sites to find new mission applications.

The segment registration form includes the following information:

- segment name
- segment prefix
- segment directory name
- list of related segments
- program management point of contact
- technical point of contact
- process point of contact
- estimated memory required by the segment
- estimated disk storage requirements
- platform availability (PC only, Solaris only, etc.)
- short paragraph describing the segment features
- list of keywords for use in catalog searches
- unclassified picture of the segment user interface (GIF, JPEG, or X11 Bitmap format).

Not all information provided at segment registration time is made available to the community at large. The *technical point of contact* is available only to the DISA Engineering Office in the event that technical questions or issues arise during segment integration. The *process point of contact* is the individual authorized by the segment program manager to actually submit the segment, or to receive status information and notifications. The *program management point of contact* is the only individual authorized to commit schedule or resources, and is the only individual authorized to release information about the segment to the community at large. The three points of contact are selected by the service/agency responsible for the segment. Services may elect to designate a single individual for all three points of contact, and may include an alternate point of contact for each category.

Each segment is assigned an identifier called a *segment prefix*. The segment prefix is a 1-6 alphanumeric character string which is used to prevent naming conflicts

between segments. Use of the segment prefix is required in any situation where there is the *possibility* that two different segment developers might choose the same name for a public symbol such as an environment variable, executable, API, or library. Two segments may in fact have the same segment prefix as long as there is no possibility that public symbols will conflict.

Segment directory names are often the same as the segment prefix, but they do not have to be. Directory names can be any directory name that conforms to rules imposed by the operating system, provided they consist only of printable characters, begin with an alphanumeric character, and are not already in use by another segment. It is recommended that directory names be limited to 14 characters to avoid porting problems.

Referring to Figure 3-1, two steps constitute the Segment Registration phase:

1. *Register the segment*. The segment registration form can be submitted in written form, through email, or in HTML format. Appendix E contains more information on how to do this. Once the developer submits the registration form, the information in entered into the CSRS and confirmation is sent to the process point of contact. Segment information is entered into the segment catalog with a tentative release date for the segment. The segment prefix and directory requested will be granted unless they have already been assigned to another developer's segment.

2. *Download segments required for development*. When notification is received that segment registration was successful, developers may download COE component segments, developer's toolkit, object code libraries, and other segments required for software development. Appendix D provides more information on how to download segments, tools, libraries, etc. It also provides information on how to access and search the on-line segment catalog.

3.2.2 Segment Development

The COE approach is designed to be non-intrusive; it places minimal constraints on how developers build, test, and manage software development. Developers are free to establish a software development environment that is best suited for their project. The COE requires only that deliveries be packaged as segments, that segments are validated before submission, and that segments are tested in the COE prior to submission. Figure 3-1 assumes this degree of freedom and omits steps such as design reviews and code walk throughs that are an expected part of any development effort.

1. *Load and configure the COE.* Most developers will find that the COE will meet their needs as is. However, for some developers the bootstrap COE may need to be extended to increase shared memory size, message queue sizes, add sockets, etc. Any changes to the downloaded COE must be carefully recorded as environment extensions. It is the responsibility of the segment to request that these extensions be made by the COE installation tools as the segment is installed.

2. *Verify that the COE is valid.* The tool `VerifyCOE` checks the integrity of the COE and should be run any time a modification is made to the bootstrap COE to ensure that the resulting environment is still COE compatible.

3. *Write and unit test code.* Develop and test a baseline version of the new software segment as independently of COE software as is possible, but within an environment as nearly identical to the actual runtime environment as is possible. The purpose of this step is to resolve problems within the segment and identify potential interface problems between the segment and the COE, especially the runtime environment. The simplest approach is to launch the segment executables from an xterm window and look for software bugs or conflicts with the COE.

4. *Create segment structures.* Chapter 5 identifies information required to describe a segment through use of segment descriptors. Decisions should be made at this point whether to package data and software together or as separate segments, how best to include any required environment extensions, how to handle segment installation and removal, which features should be icons versus menu entries, etc. Focus in the preceding step was to verify that the segment is correct *internally* while this step shifts focus to validating that the segment can interface *externally* with the COE.

5. *Validate the segment.* The tool `VerifySeg` must be run against all segments to confirm Category 1 COE compliance. `VerifySeg` must be rerun when any file within the segment that will be present at runtime is modified. This includes segment descriptor files, datafiles, and executables. A segment can not proceed any further in the process until `VerifySeg` confirms its validity. COE tools used later in the process will reject a segment that has not passed `VerifySeg`.

6. *Install and test the segment.* The tool `TestInstall` allows a segment that is already present on the disk to be installed exactly as if it had been loaded from distribution media at an operational site. When installed successfully, it should be accessible from any operator login that has a

profile set up to include the segment. At this stage, it should not be necessary to launch executables from a command line or any other interim technique. If the installation and test are not successful, the tool `TestRemove` will undo the side effects of installing the segment, but will not delete the segment from disk.

7. *Create an installation tape.* This step is optional, but recommended. The tool `MakeInstall` creates an installation tape than can then be loaded through tools in the System Administration application just as a site operator will do.

8. *Perform a system test.* Whether the segment has been installed from tape, created by `MakeInstall`, or through the `TestInstall` tool, a system level test should be performed to identify any problems with the COE or other segments for which the developer is responsible.

9. *Test segment APIs.* This step applies only to those segments, typically COE component segments, which contain APIs that other segments will use. A test suite is required for all segments which submit APIs.

3.2.3 Segment Submission

Submitting a segment is an automated process of compressing and encrypting the segment. The segment must be in the "pre-MakeInstall" format meaning that alterations made during the installation process have not been performed. These alterations are usually done by a `PostInstall` script, see Chapter 5, which may create data files, perform operations based on hardware type, etc.

1. *Compress and encrypt the segment.* The tool `mkSubmitTar` performs this task on a "pre-MakeInstall" formatted segment. The directory `Integ`, described in Chapter 5, must contain an annotated description of output from `VerifySeg.` If applicable, a test suite must be included for all APIs.

2. *Submit the segment.* The tool `submit` does this electronically across the Internet. Segments submitted via tape must be a relative tar of the output from `mkSubmitTar`, *not* the output of `MakeInstall`. Multiple segments can be delivered on the same tape provided that there is only one segment per physical tar tape segment.

3.2.4 Segment Integration

Segments received, whether by tape or electronically, are placed into the software repository (CSRS), tested in isolation, and then tested as part of the

deliverable system. Validation is performed at each step using exactly the same tool set that the developer used during the development phase. This approach allows many integration responsibilities to be performed by the developer with only a need to validate they were performed correctly when a segment reaches the traditional system integration phase.

The process steps performed from this point on in Figure 3-1 are the responsibility of DISA, not the developer. They are described here because developers are still an active part of the process in isolating and correcting problems.

1. *Receive segments.* Segments received electronically are placed in an isolated and safe disk directory. Segments received via tape are placed there manually by a member of the DISA configuration management team. The process point of contact is notified that the segment has been received and is in process.

2. *Validate the segment.* `VerifySeg` is run against the segment submitted and the results are analyzed. Discrepancies between the output of `VerifySeg` produced by the developer and that produced by the integrator can occur for a number of harmless reasons. These are reconciled against the annotated results provided by the developer when the segment was submitted. Segments which fail to pass `VerifySeg` or the reconciliation process are rejected and the process point of contact is notified.

3. *Submit segment to the CSRS.* Segments which have been validated by `VerifySeg` are compressed, encrypted, and placed in the software repository. Notification that the segment is now in the repository is sent to the process point of contact.

4. *Test segment in isolation.* The segment is loaded on a test system with the minimal segments required for the operational system. If the test fails, the process point of contact is notified with a detailed description of the problem. The segment remains in the repository but it is not available to anyone except the developer.

5. *Advance segment to test level.* Segments which work correctly in isolation are advanced to the next testing level and are so noted in the CSRS. The process point of contact is notified and developers needing the new segment are notified that a beta version is available.

6. *Define variants.* Most segments will not be loaded on every workstation. One or more variant definitions are created which contain the segment.

7. *Perform system test.* Variants containing the segment are loaded onto workstations for system testing. Those which fail are retained in the CSRS, and a list of problems are sent to the process point of contact. Depending upon the severity of the problems, the segment may be

rejected, provisionally made available for other developers to continue working, or accepted with known problems.

8. *Accept segment*. Segments which are deemed to be sufficiently stable are advanced in the test process as being ready for delivery to operational sites. This is so noted in the CSRS and notification of acceptance is sent to the process point of contact. The segment catalog is updated to reflect that the segment is now available and operational sites are notified of the new capability.

3.2.5 Segment Installation

Segments can be distributed to sites either electronically or by other distribution media as appropriate. The `MakeInstall` tool is used to extract segments from the CSRS and write them to tape or other media. The media is then manually delivered to the site. Once received at a site, the site administrator can use the installation tools in the System Administration application to load segments directly onto individual workstations. The installation tools also allow the site administrator to designate one or more workstations as *segment servers,* load segments from tape onto the segment server disk(s), and then load workstations across the site LAN from the segment servers. This greatly reduces installation time because multiple workstations can be loaded simultaneously from disk rather than serially from much slower tapes.

Installation can also be performed electronically through the `RemoteInstall` tool. The `RemoteInstall` tool operates in either a "push" or a "pull" mode. In a push mode, DISA initiates electronic transfer of segments from the CSRS to operational sites. Segments can be installed in a push mode to either a segment server or to an individual workstation. In a pull mode, the remote site initiates the segment transfer. This is done by selecting the `RemoteInstall` tool from the System Administrator application. Operating in this mode, the `RemoteInstall` tool establishes a connection to the CSRS, provides the operator with a list of segments which can be downloaded, and provides the operator with the option of loading segments onto a segment server or installing them directly onto a workstation.

3.3 Migration Considerations

The preceding section dealt with the development process as if it represents new development. However, much of the present and planned functionality is derived from existing legacy systems, not new development, and it simply is not feasible in many cases to totally abandon a system and start over. A migration strategy must be implemented which allows legacy systems to take advantage of COE benefits. The strategy must simultaneously balance full COE compliance versus implementation cost, rapid system deployment versus risk to system stability, porting functionality versus new development, and preservation of capabilities users already have versus duplication.

With the exception of subsection 3.2.2, the process outlined in the preceding section applies directly to both new development and migration strategies, or requires minimal customization. However, subsection 3.2.2, which describes the segment development phase, requires a few additional special considerations.

It is helpful to remember that the overarching approach is to build on top of the DII COE, not to decompose the COE into constituent parts to build on top of some other architecture or body of software. In other words, the approach is to integrate components *from* legacy systems *into* the COE, not to integrate the COE into an existing legacy system. This perspective is fundamental to successful integration.

The key to reusing the COE, and to achieve COE compliance, is the concept of the public API. APIs represent the gateway through which segments may gain access to COE services. Software developers and integrators must build to public APIs rather than to a particular version of the COE, since the public APIs will be preserved as the COE evolves. Applications must migrate away from private or legacy APIs since they will not necessarily be supported in subsequent COE releases.

Given this perspective of integrating components from a legacy system into the COE, the following considerations will lead to a successful migration strategy.

- *Create a requirements matrix.* The matrix should identify requirements already met by the COE, requirements that the COE meets but which require modification, and unique requirements. This matrix represents the development work which must be performed. Modifying COE functionality requires negotiation with the DISA Chief Engineer. Mission unique requirements may be met by porting legacy components, by other mission segments external to the COE, or by COTS products.

- *Develop a schedule for achieving Level 8 compliance (Full COE Compliance Level).* Compliance levels are defined in Chapter 2. Intermediate steps to achieve a lower level of compliance are very useful as progress milestones in the migration strategy. Segments must demonstrate Level 7 compliance (Interoperable Compliance) prior to acceptance as an official DISA fieldable product, and must show migration to Full COE Compliance unless the segment is targeted to be phased out.

- *Determine how the segment will be integrated with the Executive Manager.* The COE installation tools provide "hooks" to allow segment functions to be accessed as either icons from a palette or as entries from a menu pull down. The *Style Guide* contains guidelines for which approach is most appropriate for segment features.

- *Determine which account group(s) the segment will belong to.* Chapter 2 explains that account groups are used as a first order approach to dividing users into groups based on how they will use the system (system administration, database administration, etc.). This is important because it is the account group which determines the runtime environment for a segment. The COE allows a segment to belong to multiple account groups because some segments, such as a Printer segment, are of general utility while others, such as a propagation loss tactical decision aid, are much more specific to a mission application domain.

- *Determine the required runtime environment extensions.* The COE enforces the principle that segments may extend a base environment according to a set of well defined rules, but may not alter the environment in a way that adversely impacts other segments. Chapter 5 elaborates on the rules for how segments may extend the environment. The importance here is that segments *must* separate the runtime environment from software development preferences, and that identifying runtime environment deltas is the key aspect of achieving Level 3 (Workstation Compliance) compliance.

- *Identify support services within the legacy system.* These support services are candidates for replacement by COE services and should be partitioned away from the mission application through modularization of the code.

- *Identify public COE APIs to be used.* An initial step at migrating to use COE services might be to create an interim layer that maps legacy APIs to their corresponding COE APIs. This will often help in rapidly achieving Level 6 (Intermediate COE Compliance) from Level 5 (Minimal COE Compliance).

- *Negotiate new APIs or modifications with the DISA Engineering Office.* Identification of missing functionality within the COE, or with a need for modification can often serve to drive COE development.

- *Build only to public APIs.* Use of private APIs, or APIs from a legacy system, may be expedient for an interim period. However, use of such APIs will limit compliance to Level 6 or 7 and the associated risks are the responsibility of the legacy system.